MASSACHUSETTS INSTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 88-478
October 1988

EVALUATING THE PERFORMANCE OF SOFTWARE CACHE COHERENCE

Susan Owicki and Anant Agarwal

Abstract

In a shared-memory multiprocessor with private caches, cached copies of a data item must be kept consistent. This is called cache coherence. Both hardware and software coherence schemes have been proposed. Software techniques are attractive because they avoid hardware complexity and can be used with any processor-memory interconnection. This paper presents an analytical model of the performance of two software coherence schemes and, for comparison, snoopy-cache hardware. The model is validated against address traces from a bus-based multiprocessor. The behavior of the coherence schemes under various workloads is compared, and their sensitivity to variations in workload parameters is assessed. The analysis shows that the performance of software schemes is critically determined by certain parameters of the workload: the proportion of data accesses, the fraction of shared references, and the number of times a shared block is accessed before it is purged from the cache. Snoopy caches are more resilient to variations in these parameters. Thus when evaluating a software scheme as a design alternative, it is essential to consider the characteristics of the expected workload. The performance of the two software schemes with a multistage interconnection network is also evaluated, and it is determined that both scale well.

Keywords: computer software, computer hardware; (KR)

Microsystems
Research Center
Room 39-321

Massachusetts
Institute
of Technology

Cambridge
Massachusetts
02139

Telephone
(617) 253-8138

Acknowledgements

To appear in Proceedings, Architectural Support for Programming Languages and Operating Systems, March 1989. This work was supported in part by the Defense Advanced Research Projects Agency under contract no. N00014-87-K-0825.

Author Information

Owicki: Systems Research Center, Digital Equipment Corporation, Palo Alto, CA.

Agarwal: Laboratory for Computer Science and the Department of Electrical Engineering and Computer Science, Room NE43-418, MIT, Cambridge, MA 02139, (617) 253-1448.

Evaluating the Performance of Software Cache Coherence

Susan Owicki
Systems Research Center
Digital Equipment Corporation
Palo Alto, CA

Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA

September 29, 1988

Abstract

In a shared-memory multiprocessor with private caches, cached copies of a data item must be kept consistent. This is called cache coherence. Both hardware and software coherence schemes have been proposed. Software techniques are attractive because they avoid hardware complexity and can be used with any processor-memory interconnection. This paper presents an analytical model of the performance of two software coherence schemes and, for comparison, snoopy-cache hardware. The model is validated against address traces from a bus-based multiprocessor. The behavior of the coherence schemes under various workloads is compared, and their sensitivity to variations in workload parameters is assessed. The analysis shows that the performance of software schemes is critically determined by certain parameters of the workload: the proportion of data accesses, the fraction of shared references, and the number of times a shared block is accessed before it is purged from the cache. Snoopy caches are more resilient to variations in these parameters. Thus when evaluating a software scheme as a design alternative, it is essential to consider the characteristics of the expected workload. The performance of the two software schemes with a multistage interconnection network is also evaluated, and it is determined that both scale well.

1 Introduction

Shared-memory multiprocessors often use per-processor caches to reduce memory latency and to avoid contention on the network between the processors and main memory. In such a system there must be some mechanism to ensure that two processors reading the same address from their caches will see the same value. Most schemes for maintaining this *cache coherence* use one of three approaches: snoopy caches, directories, or software techniques.

Snoopy cache methods [11, 14, 17, 21, 30] are the most commonly used. A snoopy cache-controller listens to transactions between main memory and the other caches and updates its state based on what it hears. The nature of the update varies from one snoopy-cache scheme to another. For example, on hearing that some cache has modified the value of a block, the other caches could either invalidate or update their own copy. Because all caches in the system must observe memory transactions, a shared bus is the typical medium of communication.

Another class of techniques associates a directory entry with each block of main memory; the entry records the current location(s) of each memory block [29, 4]. Memory operations query the directory to determine whether cache-coherence actions are necessary. Directory schemes can be used with an arbitrary interconnection network.

Both snoopy cache and directory schemes involve increased hardware complexity. However, the caches are invisible at the software level, which greatly simplifies programming these machines. As an alternative, cache coherence can be enforced in software, trading software complexity for hardware complexity. Software schemes have been proposed by Smith [28] and Cytron [7] and are part of the design or implementation of the Elxsi System 6400 [23, 22], NYU Ultracomputer [8], IBM RP3 [3, 10], Cedar [5], and VMP [6].

Software schemes are attractive not only because they require minimal hardware support, but also because they can scale beyond the limits imposed by a bus. We will examine two sorts of software schemes in this paper. The simplest approach is to prohibit caching of shared blocks. Shared variables are identified by the programmer or the compiler. They are stored in regions that are marked as non-cachable, typically by a tag or a bit in the page table. Loads and stores to those regions bypass the cache and go directly to main memory, while references to non-shared variables are satisfied in the cache. Such a scheme was used in C.mmp [12] and the Elxsi System 6400 [23, 22]. We refer to this approach as the *No-Cache* scheme.

In another software approach, which we will call *Software-Flush*, shared variables can be removed from the cache by explicit flush instructions. These instructions may be placed in the program by the compiler or the programmer. A typical pattern is to operate on a set of shared variables within a critical section and to flush them before leaving the critical section. This will force any modified variables to be written back to memory. Then the next reference to a shared variable in any processor will cause a miss that fetches the variable from memory. A more sophisticated scheme might allow multiple read copies of blocks, and have processes explicitly synchronise and flush cache blocks when performing a write.[1] If the flush instructions are to be inserted by the compiler, it must be possible to detect shared variables and the boundaries of execution within which a shared variable can remain in the local cache. Such regions can be made explicit in the language or detected by compile-time analysis of programs [7]. Except when there is very little shared data, good performance from the Software-Flush scheme places considerable demands on the compiler.

This paper analyses the performance of the No-Cache and Software-Flush schemes. For comparison, we also examine a snoopy-cache scheme, which we call *Dragon*, and the *Base* scheme, which does not take any action to preserve coherence. The questions we address include: What sort of performance can we expect from such schemes? How is their performance effected by scaling? Are there differences in performance between systems based on a bus and a multistage interconnection network? How do variations in the workload affect performance?

We define an analytical multiprocessor cache model, and use it to predict the overhead of the four cache-coherence schemes over a wide range of workload parameters. We chose this approach, rather than simulation, for several reasons. Trace-based simulation was impossible due to the lack of suitable traces. Simulation with a synthetic workload was possible, and would have allowed us to model more detailed features of the coherence schemes. However, there seems to be little benefit in doing this; we can see significant variation among the schemes even without this detail. Evaluating the analytic model is much faster than performing either type of simulation, which allows us to study the schemes over a wide range of workload parameters. This is especially useful for software schemes, where there is as yet little workload data. However, because the results of analytical models are always subject to doubt, we have validated our model against simulation with real address traces from a small multiprocessor system.

We observed that the performance of the software schemes was most affected by the frequency of data references, degree of sharing, and number of references to a shared datum between fetching and flushing. These parameters impact the performance of software schemes much more dramatically than the Dragon scheme. Software caching works well in favorable regions of the above parameters, but does badly in other regions. Therefore, it is critical to estimate the expected range of these parameters when evaluating a software scheme.

Both the Software-Flush scheme and the No-Cache scheme scale to systems with general memory interconnection networks. In such a system, the efficiency of the Software-Flush scheme drops heavily when the workload is heavy, while the efficiency of the No-Cache scheme becomes abysmal even with moderate workload.

[1] Some schemes even allow temporary inconsistency to reduce serialization penalties [7].

Previous cache-coherence studies have focused on the performance of hardware-based schemes. Archibald and Baer [2] evaluate a number of snoopy-cache schemes using simulation from a synthetic workload. A similar analysis using timed Petri nets was performed by Vernon et al. [31]. A mean value analysis of snoopy cache coherence protocols was presented by Vernon, Lasowska and Zahorjan [32]. They were able to achieve very good agreement between the earlier Petri net simulation and an analytic model that was much less computationally demanding. The approach taken in this last paper is the closest to ours. Greenberg and Mitrani [15] use a slightly different probabilistic model to analyse several snoopy cache protocols. A model characterising multiprocessor references and their impact on snoopy schemes is presented by Eggers and Katz [9]. Directory schemes are evaluated by Agarwal et al. [1] using simulation with multiprocessor address traces.

The rest of the paper is organized as follows: Section 2 presents the cache model for bus-based multiprocessors, and the following section describes its validation. Section 4 performs a sensitivity analysis to determine the critical parameters in the various schemes. The results of the analyses for buses are presented in Sections 5. Section 6 gives the model and analysis of a multiprocessor with a multistage interconnection network.

2 The Model

We wish to compare the cost of cache activity in the No-Cache, Software-Flush, Dragon, and Base schemes. Cache overhead consists of the time spent in handling cache misses and implementing cache coherence. Processor utilization U is the fraction of time spent in "productive" (non-overhead) computation. An n-processor machine has ***processing power*** $n \times U$, and we use processing power as the basis for our comparisons.

An analytic model is used to estimate processing power. It has three components. The ***system model*** defines the cost of the operations provided by the hardware. The ***workload model*** gives the frequency with which these operations are invoked, expressed in terms of various workload parameters. From these two models it is possible to determine the average processor and bus time required by an instruction. Additional time is lost to contention for the shared bus or the interconnection network, and this is estimated by the ***contention model***. Only the bus model is defined in this section; the network model is defined in Section 6.2.

2.1 System Model

Table 1 lists the operations in the model. In addition to instruction execution, clean miss, and dirty miss, they include specific operations for each coherence scheme. For No-Cache, there are read-through and write-through operations to access a word in main memory rather than a word in the cache. For Software-Flush, a flush instruction invalidates a block in the cache and writes the block back to main memory if it is dirty. Finally, the Dragon scheme has write-broadcast, a miss (clean or dirty) satisfied from another cache, and cycle-stealing by the cache controller.

Table 1 also gives the CPU and bus time, in cycles, for each operation. Here CPU time is the total time required for the operation in the absence of contention, and bus time is the part of that time during which the bus is held. (Bus and CPU cycle time are assumed to be the same.)

The operation costs are based on a hypothetical RISC machine with a combined instruction and data cache. Each instruction takes 1 cycle, plus the time for any cache operations it triggers. The cost of cache operations is based on a block size of four words. Thus, for example, a load which causes a clean miss from memory needs 7 cycles of bus time, one to send the address, two for memory access, and four to get the 4 words of data. Processor time to detect and process the miss adds 3 CPU cycles for a total of 10. Finally the load itself is performed, adding one more CPU cycle for instruction execution. The times for other instructions are derived in a similar way.

Operation	CPU Time	Bus Time
Instruction execution (except flush)	1	0
Clean miss (mem)	10	7
Dirty miss (mem)	14	11
Read through	5	4
Write through	2	1
Clean flush	1	0
Dirty flush	6	4
Write broadcast	2	1
Clean miss (cache)	9	6
Dirty miss (cache)	13	10
Cycle stealing	1	0

Table 1: System model: CPU and bus time for hardware operations

2.2 Workload Model

The workload model determines the frequency of the operations defined in the system model. The operation frequencies are expressed in terms of the parameters listed in Table 2. The "shared data" in this table means slightly different things in the software and Dragon schemes. For No-Cache and Software-Flush, an item is *shared* if it is treated as shared by the cache coherence algorithm; this is determined by the compiler or programmer. For the Dragon scheme, an item is *shared* if it is actually referenced by more than one processor. These interpretations should not lead to widely differing values.

	For all schemes
ls	probability an instruction is a load or store
msdat	miss rate for data
msins	miss rate for instructions
md	probability a miss replaces a dirty block
shd	probability a load or store refers to shared data
wr	probability a miss is caused by store rather than load
	For Software-Flush only
apl	number of references to a shared block before it is flushed
mdshd	probability a shared block is modified before it is flushed
	For Dragon only
oclean	on miss of a shared block in one cache, probability it is not dirty in another
opres	on reference to a shared block in one cache, probability it is present in another
nshd	on write-broadcast, number of caches containing a shared block

Table 2: Parameters for the Workload Model

Some of these parameters are functions of the underlying system as well as of the program workload. For example, miss rates depend on block size, cache size, and so on. We don't try to model those effects, since they are not relevant to cache coherence. It is enough to consider a range of values for those parameters.

The remainder of this section describes the workloads of the four cache-coherence schemes. The information here, combined with the system model, makes it possible to compute the average rate and service time of bus transactions. Let o denote a hardware operation, $freq(o, scheme)$ the frequency of that operation in the workload model for $scheme$, $cycles(o, cpu)$ the CPU time for o, and $cycles(o, bus)$

the bus time for o in the system model. Then an instruction takes an average of

$$c = \sum_o freq(o, scheme) \times cycles(o, cpu) \tag{1}$$

CPU cycles and

$$b = \sum_o freq(o, scheme) \times cycles(o, bus) \tag{2}$$

bus cycles. Thus b is the average transaction service time, and $1/(c-b)$ is the average transaction rate. In other words, bus transactions are generated at an average rate of one every $c-b$ CPU cycles, and each transaction requires an average of b bus cycles.

2.2.1 Base

The Base scheme, which does not implement coherence, is included to give an upper bound on performance. Its workload is characterized in Table 3.

clean miss	$(ls \times msdat + msins) \times (1 - md)$
dirty miss	$(ls \times msdat + msins) \times md$

Table 3: Workload model: Base scheme

Some discussion of these formulae may be useful. A data miss occurs when a load or store instruction (probability ls) refers to an address that is not present in the cache (probability $msdat$). An instruction miss occurs with probability $msins$. In either case, if the block to be replaced is dirty (probability md) the miss is dirty, if not (probability $1 - md$) it is clean.

2.2.2 No-Cache

In this scheme, shared variables are identified by the programmer or the compiler. They are stored in memory regions that are marked as non-cachable, typically by a tag or a bit in the page table. Loads and stores to those regions bypass the cache and go directly to main memory.

Table 4 gives the frequencies of cache operations for the No-Cache scheme. The probability of a data miss is reduced from the Base scheme by a factor of $1 - shd$, because only unshared data is kept in the cache. In addition, all loads (stores) to shared variables require a read-through (write-through) operation.

clean miss	$(ls \times msdat \times (1 - shd) + msins) \times (1 - md)$
dirty miss	$(ls \times msdat \times (1 - shd) + msins) \times md$
read-through	$ls \times shd \times (1 - wr)$
write-through	$ls \times shd \times wr$

Table 4: Workload model: No-Cache

2.2.3 Software-Flush

In this approach, shared variables can be removed from the cache by explicit flush instructions. These instructions may be placed in the program by the compiler or the programmer. A typical pattern is to operate on a set of shared variables within a critical section, and to flush them before leaving the critical section. This will force any variables modified in the critical section to be written back to memory. Then the first reference to a shared variable within the next critical section will cause a miss that fetches the variable from memory. If the flush instructions are to be inserted by the compiler, it must be possible to detect shared variables and the boundaries of execution within which a shared variable can remain locally cached. Such regions can be made explicit in the language or detected by compile-time analysis of programs [7].

Table 5 gives the frequency of operations for Software-Flush. Non-shared variables generate the same number of clean and dirty misses as in the No-Cache scheme. Shared variables are handled by inserting flush instructions at an average rate of one per *apl* references to shared variables, i.e. one per $\mathit{apl} \times \mathit{shd} \times \mathit{ls}$ instructions. This increases the operation frequency in three ways. First, there is the cost of the flush instruction itself; with probability *mdshd* this will be a dirty flush. Second, there is approximately one clean miss for each flush instruction, namely, the miss which brought the flushed line into the cache. This approximation assumes that the probability of the line's being replaced in the cache before it is flushed is low enough to be ignored. Finally, the added flush instructions increase the instruction miss rate. Note that Table 5 reports operation frequencies per non-flush instruction. This is because flush instructions are part of the cache-coherence overhead, and their cost is amortised over the other instructions.

clean miss	$(\mathit{ls} \times \mathit{msdat} \times (1 - \mathit{shd}) + \mathit{msins}) \times (1 - \mathit{md})$ $+(\mathit{ls} \times \mathit{shd}/\mathit{apl}) \times \mathit{msins}$
dirty miss	$(\mathit{ls} \times \mathit{msdat} \times (1 - \mathit{shd}) + \mathit{msins}) \times \mathit{md}$
clean flush	$\mathit{ls} \times \mathit{shd} \times (1 - \mathit{mdshd})/\mathit{apl}$
dirty flush	$\mathit{ls} \times \mathit{shd} \times \mathit{mdshd}/\mathit{apl}$

Table 5: Workload model: Software-Flush

Both No-Cache and Software-Flush may be available on the same machine. On the Elxsi 6400 [23, 22], the programmer determines whether a particular shared variable is kept coherent by the No-Cache or Software-Flush scheme. In the MultiTitan [16], locks are not cached, and other shared variables are kept coherent by Software-Flush.

2.2.4 Dragon

We have modeled one snoopy bus protocol to provide a comparison point for the software mechanisms. A Dragon-like scheme [21] was selected because Archibald and Baer [2] found its performance to be among the best.

The following is a slightly simplified description of the relevant aspects of the Dragon protocol. From listening to bus traffic, a cache knows if an address is valid in another cache. When a store refers to an address that is in another cache, the address and new value are broadcast on the bus, and any cache that has the address updates its value accordingly. All stores to non-shared addresses are performed in the local cache. On a cache miss, main memory supplies the block unless it is dirty in another cache; in the latter case, that cache supplies the block.

Table 6 gives the frequency of operations for the Dragon model. There are three effects to consider. First, the write-broadcast occurs once per $\mathit{shd} \times \mathit{opres}$ writes. Second, some misses will be satisfied from a cache instead of from main memory; this happens on a miss with probability $\mathit{shd} \times (1 - \mathit{oclean})$. Finally, a write-broadcast may cause other caches to steal cycles from their processors as they update their own copy of the variable. This occurs with frequency *nshd* on each write-broadcast. As it happens, the last two effects are small and could have been omitted from the model without significantly affecting our results.

clean miss from memory	$\mathit{ls} \times \mathit{msdat} \times (1 - \mathit{shd} \times (1 - \mathit{oclean}) + \mathit{msins}) \times (1 - \mathit{md})$
dirty miss from memory	$\mathit{ls} \times \mathit{msdat} \times (1 - \mathit{shd} \times (1 - \mathit{oclean}) + \mathit{msins}) \times \mathit{md}$
write broadcast	$\mathit{ls} \times \mathit{shd} \times \mathit{wr} \times \mathit{opres}$
clean miss from cache	$\mathit{ls} \times \mathit{msdat} \times \mathit{shd} \times (1 - \mathit{oclean}) \times (1 - \mathit{md})$
dirty miss from cache	$\mathit{ls} \times \mathit{msdat} \times \mathit{shd} \times (1 - \mathit{oclean}) \times \mathit{md}$
steal cycle	$\mathit{ls} \times \mathit{shd} \times \mathit{wr} \times \mathit{opres} \times \mathit{nshd}$

Table 6: Workload model: Dragon

2.3 Contention Model

An n-processor system can be modeled as a closed queueing network with a single server (the bus) and n customers (the processors). Such a network is characterized by two parameters: the average service time and average rate of bus transactions.[2] In our system, the average service time is b cycles and the average rate is $1/(c-b)$ transactions per cycle, where c and b are defined in equations 1 and 2 respectively. Solution of the queueing model [20] yields w, the contention cycles per instruction. Thus the total time per instruction is $c+w$. In the absence of cache activity, an instruction would take 1 cycle, so the CPU utilization is

$$U = 1/(c+w) \tag{3}$$

This contention model is very similar to the one used by Vernon et al. [32] in analyzing snoopy-cache protocols.

3 Validation

This section compares model predictions against simulation results for the Base scheme and Dragon schemes. We developed a trace-driven multiprocessor cache and bus simulator, which can compute statistics like cache miss rates, cycles lost to bus contention, and processor utilization, for a variety of coherence schemes, cache sizes and processor numbers.

The address traces used in the validation were obtained using ATUM-2, a multiprocessor tracing technique described in [26]. The traces contain interleaved memory references from the processors. Three of the traces (POPS, THOR, and PERO) were taken on a four-processor VAX 8350 running the MACH operating system. We also used an 8-processor trace of PERO, which was obtained from a parallel tracer using the VAX T-bit. The four-processor traces include operating system references, and none of the traces include process migration. Sites and Agarwal [26] describe the applications and details of the traces.

We simulated 16K, 64K, and 256K-byte caches with a fixed blocksize of 16 bytes and the same transfer block size. The hardware model used is summarized in Table 1. The model was validated only for the Base and the Dragon schemes. Meaningfully validating the software schemes was not possible because the traces are from a multiprocessor that used hardware for cache coherence. Because the multiprocessor model used in the simulations is different from the traced machine model, the order of references from different processors may be slightly distorted in the simulation. However, we expect that this effect is not large, because the timing differences between the two multiprocessor models affect the address streams from all the processors uniformly. Also, the cache statistics we obtained matched those from simulating a multiprocessor model that retained the exact order of the references in the trace.

For a multiprocessor cache model to be useful, it is important that the parameter values input to the model are valid over the range in which the model is exercised. Variations in input parameters may be modeled, or a parameter value must be input for each point under consideration. In general, we choose parameters that are expected to be nearly constant as the number of processors increases, and verify that they are nearly constant in the trace-driven simulations. The parameters of concern in our model are *oclean* and *opres*, which can vary with the number of processors in a way that depends on program structure. In our traces, we did observe some random variations in these parameters, but they were small enough that the model was still accurate. A comparative evaluation of snoopy caches should somehow account for the variations in *oclean* and *opres*.[3] But our focus is on software cache coherence, and we can safely ignore this issue.

The model results closely match simulations in most cases. Figures 1 through 3 present a sampling of our experiments comparing the model predictions to simulations. We will address potential sources of inaccuracies in the ensuing discussion.

Figure 1 shows the results for 64K-byte caches for the Base and Dragon schemes. While the model

[2] If a multistage interconnection network is used, the multistage network is represented as a load-dependent service center characterised by its service rate at various loads.

[3] For invalidation-based snoopy caches, the miss rate also falls in this category.

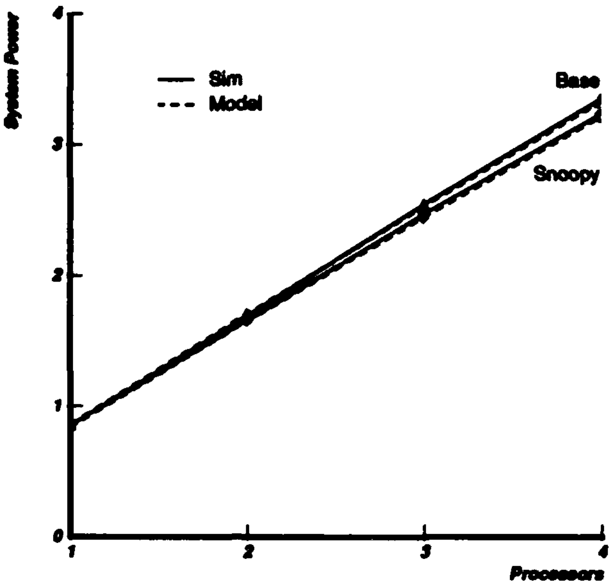

Figure 1: A performance comparison of the Base and the Dragon schemes using simulation and the analytical model for 64K-byte caches.

exactly captures the relative difference between the performance of Base and Dragon schemes, it consistently overestimates bus contention. This is because the bus model is based on exponential service times, while the simulations use fixed bus service times for the different bus operations.

Figure 2 shows the model and simulation results for three cache sizes for the Dragon scheme. Minor inconsistencies between the model and simulation results for single processors can be attributed to the difference in the values of *oclean* and *opres* for one and four processors. The model shows similar accuracy in the 8-processor plot depicted in Figure 3.

4 Sensitivity Analysis

The workload model uses a number of parameters to characterize the program's workload. One would expect some of them to be quite important, and others to have little impact on cache performance. This section describes the sensitivity analysis that we used to estimate the importance of each parameter.

The significance of a parameter was assessed from the change in execution time when that parameter was varied and all others were held constant. We chose low, middle, and high values for each parameter, representing the range of values likely to be seen in programs. The ranges are given in Table 7. They were derived from the minimum, average, and maximum values observed in the large-cache traces, except as follows.

There was not enough data in the traces to determine *apl*, so it was estimated by counting the number of references of a cache-line by one processor (at least one of which was a write) between references by another processor. This is an optimistic estimate, so the upper bound of $1/apl$ was taken to be 1, the maximum possible. The values of *md* from the trace were artificially low, because the traces were not long enough to fill up the large caches. The measured high value was 0.25, but 0.5 was used as the high value in the sensitivity analysis; values of this magnitude have been measured by Smith [27]. Finally, the range for *ls* is typical for RISC architectures rather than the CISC machine on which the traces were

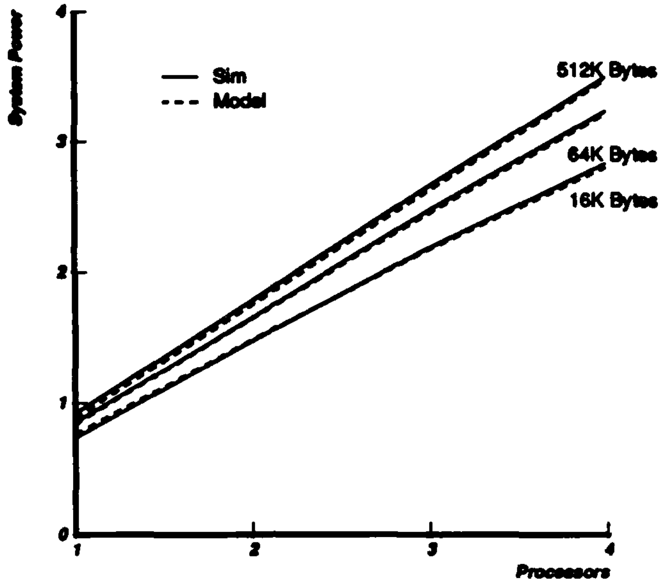

Figure 2: Impact of the cache size on the performance of the Dragon scheme for four or fewer processors.

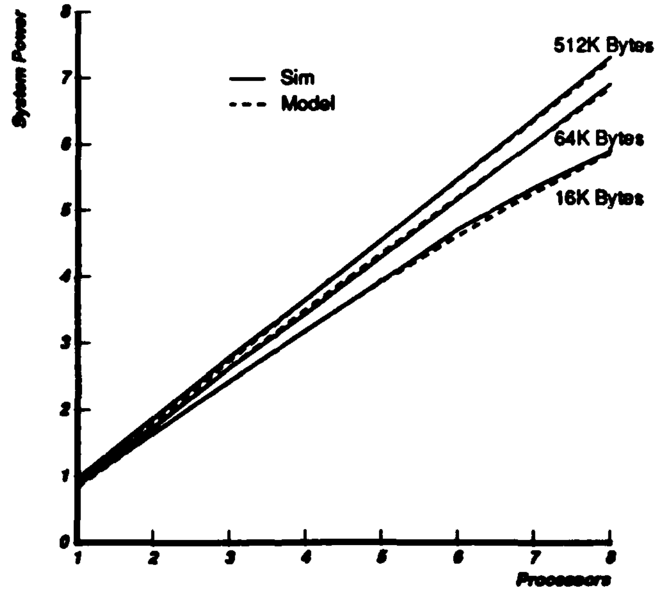

Figure 3: Impact of the cache size on the performance of the Dragon scheme for eight or fewer processors.

taken.

Parameter	Low	Middle	High
ls	0.2	0.3	0.4
msdat	0.004	0.014	0.024
msins	0.0014	0.0022	0.0034
md	0.14	0.20	0.50
shd	0.08	0.25	0.42
wr	0.10	0.25	0.40
mdshd	0.0	0.25	0.5
1/apl	0.04	0.13	1.0
oclean	0.60	0.84	0.976
opres	0.63	0.79	0.94
nshd	1.0	1.0	7.0

Table 7: **Parameter ranges**

Table 8 shows the results of the sensitivity analysis. For each parameter, we computed the per cent change in execution time when the parameter changes from low to high, with all other parameters held constant. (Note that in all cases execution time is greater for the low value of the parameter.) This computation was performed for three settings of the other parameters: low, middle, and high. The maximum per cent change is reported in the table.

Base		No-Cache		Software-Flush		Dragon	
msdat	17	shd	65	1/apl	88	ls	19
ls	11	ls	48	shd	74	msdat	17
msins	4	msdat	10	ls	49	shd	11
md	4	msins	4	msdat	10	opr	4
		md	1	mdshd	4	msins	4
		wr	<1	msins	4	md	4
				md	1	nshd	4
						wr	3
						oclean	<1

Table 8: **Sensitivity to parameter variation, depicted by the per cent change in execution time when the parameter changes from low to high, with all other parameters held constant.**

The numbers from the sensitivity analysis must be interpreted with care. The choice of range affects how important a parameter appears. For example, our traces show a small variation in miss rates, and miss rate shows only a modest effect in the sensitivity analysis. Had our traces exhibited greater variation in miss rate, it would have appeared to be much more significant. In addition, changing the miss rate range can change the apparent significance of other parameters, because their effect is estimated at high, low and middle values of miss rate. A wide range may represent a parameter that is observed to vary widely in practice (e.g. *shd*) or a parameter about which we have little information (e.g. *apl*).

In spite of these caveats, certain parameters are clearly more important than others. For the Software-Flush scheme, *apl* has a huge effect; this is due to both its central importance in the scheme and its wide range. The impact of *shd* is almost as great, and *ls* is significant as well. Miss rate has a noticeably smaller effect, and the other parameters are relatively unimportant. The No-Cache scheme is essentially the same, except that *apl* is not relevant. In the Dragon scheme, the overall hit rate is more important than the level of sharing, even though its range is quite small, because the cost of shared references is relatively low.

In the next section we will analyze the effect of *apl*, *ls* and *shd* in more detail. The effect of *ls* is primarily as a multiplier of *shd* and *msdat*; so the analyses of *ls* and *shd* will be combined by varying them jointly. Parameters *mdshd* and *wr*, which are specific to the Software-Flush and No-Cache schemes, were examined further in spite of their low showing in the sensitivity analysis. When allowed to vary

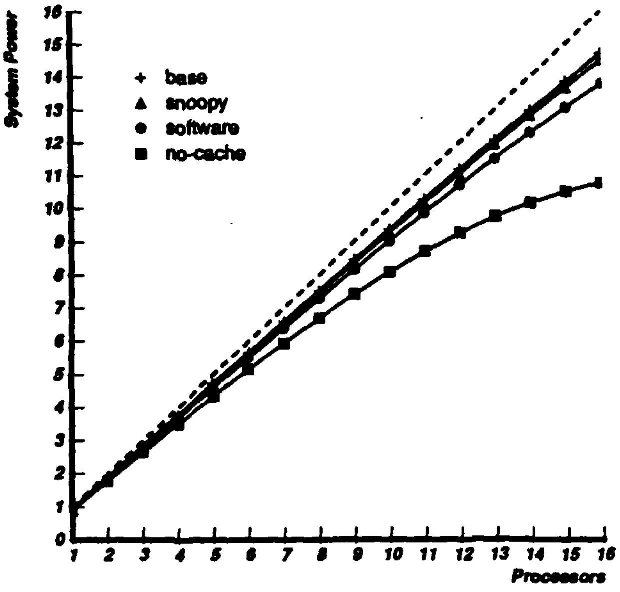

Figure 4: Performance of cache-coherence schemes with low *shd* and *ls*; all other parameters at medium values

over a wider range, *mdshd* had a small but noticeable effect on the Software-Flush scheme; but *wr* was unimportant even with a wide range.

5 Bus Performance

5.1 Variations among Coherence Schemes

Figures 4 through 6 show the relative performance of the four cache coherence schemes for three settings of *ls* and *shd*. The dotted line is the theoretical upper bound on processing power. It represents the case in which each processor is fully utilized, and there is no delay due to memory activity. All schemes fall below this line, because a processor is delayed when it uses the bus in handling cache misses and references to shared data. With multiple processors the cost of bus operations increases because contention can add a significant delay. For this reason, the incremental benefit of adding a processor is smaller for large systems than small ones.

Comparing the schemes, we find that Base performs best as long as $ls > 0$; this is to be expected, since the others incur overhead in processing shared data. (If $ls = 0$ the schemes are identical.) In most cases Dragon's performance is close to Base. It incurs sharing overhead only when data are simultaneously in the caches of two or more processors, and then only on write operations, i.e. once every $shd \times opr \times wr$ references. Moreover, the overhead is relatively small, since only one word needs to be transmitted on the bus. No-Cache is much more costly than Dragon, because the processor must go to main memory on every reference to a potentially-shared variable, i. e. once very *shd* references. Software-Flush's performance is drastically affected by the value of *apl*, because there is a main memory operation on every $1/apl$ references to shared data. As will be illustrated in Section 5.3, Software-Flush's performance is usually between Dragon and No-Cache, but it can be better than Dragon or worse than No-Cache.

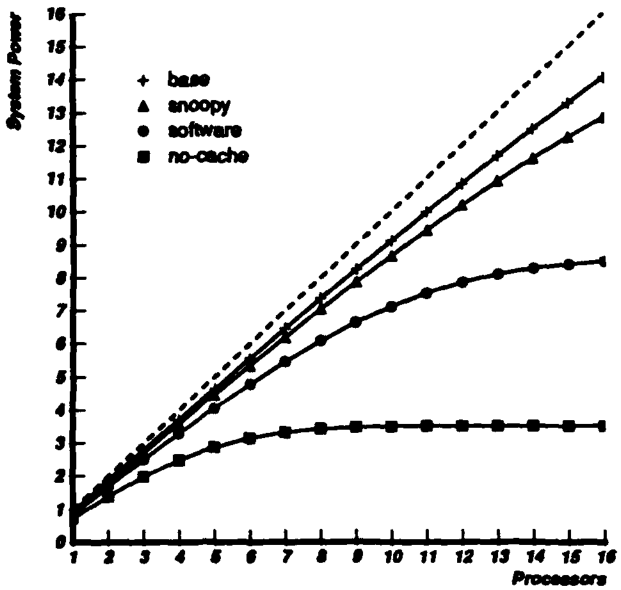

Figure 5: Performance of cache-coherence schemes with medium *shd* and *ls*; all other parameters at medium values

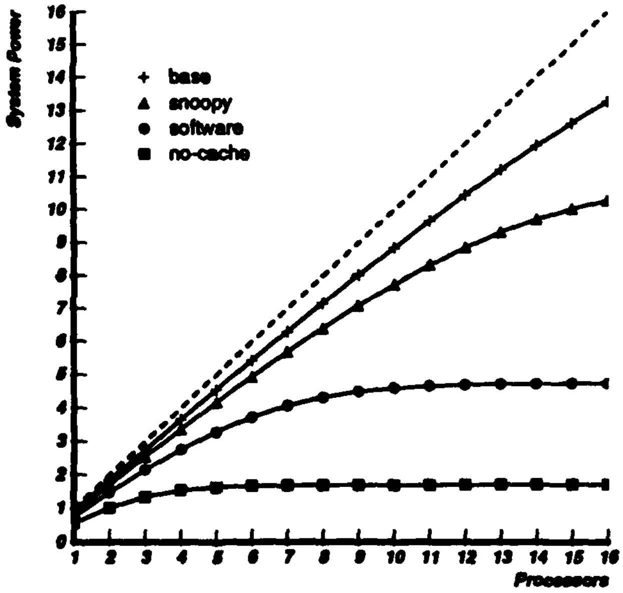

Figure 6: Performance of cache-coherence schemes with high *shd* and *ls*; all other parameters at medium values

5.2 Effect of *ls* and *shd*

Parameter *ls* has a significant impact on all schemes, and *shd* is important for all but Base. Both affect the frequency of memory activity. *ls* determines the frequency of data references in the instruction stream, while *shd* determines the proportion that go to shared data items. Thus increasing *ls* has a double effect on overhead: it increases both the frequency of misses and the frequency of shared data references.

At low values of *ls* and *shd* (Figure 4), Base, Dragon, and Software-Flush perform well, and there is not much difference between them. (Recall that the Software-Flush scheme is evaluated with a medium *apl* value.) Even No-Cache performs well for a moderate number of processors. Low levels of sharing can be expected in some situations: for example, if a multiprocessor is used as a time sharing system, so that separate processors run unrelated jobs, or if communication is done through messages rather than shared memory [23]. In such environments No-Cache is a viable alternative.

Even with middle values of *ls* and *shd* (Figure 5), No-Cache performs acceptably with a small number of processors. Dragon performs very well even with 16 processors. With medium *apl*, Software-Flush does well with up to 8-10 processors; from then on, adding processors only slightly increases processing power.

With high *ls* and *shd* (Figure 6), Dragon still gives good performance. No-Cache does badly; it saturates the bus with a processing power less than 2. Software-Flush performs acceptably for a small number of processors; it saturates the bus with processing power less than 5. Even in this high sharing region, however, Software-Flush can perform well if *apl* is high.

5.3 Effect of *apl*

The performance of Software-Flush is drastically affected by the value of *apl*. Figure 7 illustrates the variation that can occur. When $apl = 1$, every reference to a shared variable requires a flush (possibly dirty) and a miss. This means that both CPU and bus demands are heavier than for No-Cache, and, indeed, Software-Flush's performance is the worse. On the other hand, very high values of apl make sharing overhead very small, especially if *mdshd* is not low. In this case Software-Flush can perform as well as Dragon, or even better.

Figures 8 and 9 show the variation of processing power with *apl* for two levels of sharing. With low sharing, performance is very sensitive to *apl* at low values, but quickly reaches its maximum as apl is increased. With medium sharing levels, performance is sensitive to variations in *apl* even at relatively high values.

The range for *apl* reported from our traces is optimistic: it assumes that data is flushed only when absolutely necessary. As our measurements show, the number of uninterrupted references to a shared-written object by a processor can be quite large in practice. It remains to be seen whether a compiler can generate code that takes advantage of these long runs. Doing so is crucial if software schemes are to be used in the presence of even moderate amounts of shared data.

6 Interconnection Network Performance

Unlike the snoopy schemes, the software schemes can be used in a network environment where there is no mechanism for a cache to observe all the processor-memory traffic. In this section we explore the scalability of software schemes in such an environment. Some of the questions we address are: Is caching shared data in a network environment worthwhile? Can software schemes scale to a large number of processors?

The analysis uses a multistage interconnection network model to evaluate the system performance of a cache-based multiprocessor. The workload model is the same as before, and new models for hardware timing and network contention are discussed in the next sections. As in our bus analysis, we first compute the average transaction rate and transaction time for the network, then use the contention model to compute the network delay. From these, system processing power can be computed as before.

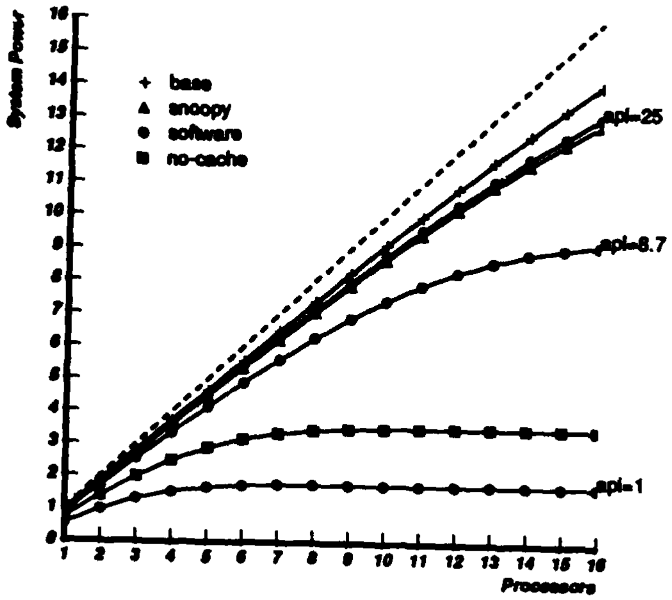

Figure 7: Effect of varying *apl*; other parameters at medium values

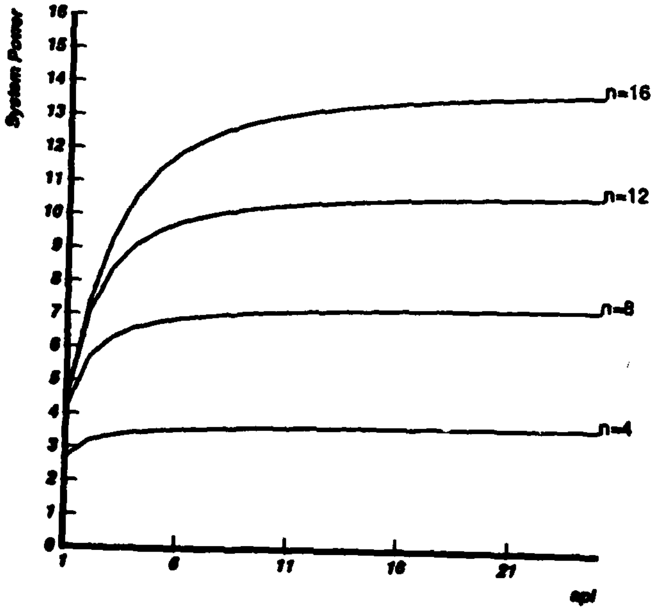

Figure 8: Effect of *apl* with low sharing; other parameters at medium values

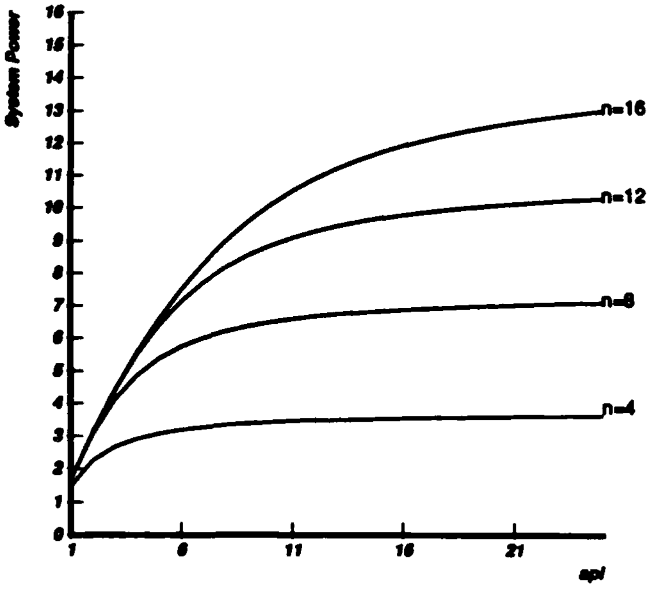

Figure 9: Effect of *spl* with medium sharing; other parameters at medium values

6.1 The Network

Our analysis applies to the general class of multistage interconnection networks called Banyan [13], Omega [19], or Delta [25]. For our analysis we consider an unbuffered, circuit-switched network composed of 2x2 crossbars, with unit dilation factor. The analysis can be extended easily to dilated networks or crossbar switches with a larger dimension. A request accepted by the network travels through n switch stages (corresponding to a system with 2^n processors) to the memory; the response from the memory returns on the path established by the request. If multiple messages are simultaneously routed to the same output port, a randomly-chosen one is forwarded and the other is dropped. The source is responsible for retransmitting dropped messages. A switch cycle is assumed to be the same as a processor cycle. The network paths are assumed to be one word (4 bytes) wide,[4] and a cache block is 4 words long, as before.

We have tried to keep the network timing model consistent with the bus where possible. Table 9 gives the network timing model. The network delay (without contention) for a cache miss is $6 + 2n$ cycles, n to set up the path, 1 to send the address to memory, 2 for memory access, n for the return of the first data word, and 3 for the remaining 3 words. (We will sometimes refer to the network service time minus $2n$ as the message size.) Similarly, a dirty miss costs $9 + 2n$ cycles, n to set up the path, 1 to send the address to memory, 2 for memory access (overlapped with getting the address of the dirty block and one data word), 3 cycles for the remaining dirty words, n for the return of the first word, and 3 for the remaining 3 words.

6.2 The Network Contention Model

Our network analysis uses the model due to Patel [24]. Patel's model has been used extensively in the literature. We are not aware of any validation of this model against multiprocessor traces, although is

[4] The wide path is one of the reasons we use 2x2 switches, because larger dimension switches will not fit easily into a single chip with current technology.

Operation	CPU Time	Network Time
Instruction execution		
(Except flush)	1	0
Clean fetch	$9+2n$	$6+2n$
Dirty fetch	$12+2n$	$9+2n$
Clean flush	1	0
Dirty flush	$7+2n$	$5+2n$
Write through	$3+2n$	$2+2n$
Read through	$4+2n$	$3+2n$

Table 9: System model for a network with *n* stages

has been tested using simulations based on synthetic reference patterns.[5]

Our analysis requires certain assumptions for tractability; they are similar to the ones typically made in the literature [24, 18]. We assume that the requests are independent and uniformly distributed over all the memory modules. An average transaction rate m and an average transaction size t is computed for each of the cache coherence schemes; these correspond to $1/(c-b)$ and b from equations 1 and 2 in the bus analysis. The network delay can be estimated using the unit-request approximation, in which the transaction rate is taken to be mt and the transaction size to be 1. It is as if the processor splits up a t-unit memory request into t independent and uniformly distributed unit-time sub-requests. Patel validates the accuracy of this approximation through simulation.

Let m_i be the probability of a request at a particular input at the n^{th} stage of the network in any given cycle. Then, the effective processor utilization U, and hence system processing power, can be computed using the system of equations given below. The value of m_n in the equations is obtained recursively for successive stages starting with the input request rate of m_0. The equations can be solved using standard numerical techniques.

$$\begin{aligned} U &= \frac{m_n}{mt} \\ m_{i+1} &= 1-(1-\frac{m_i}{2})^2 \quad 0 \leq i < n \\ m_0 &= 1-U \end{aligned}$$

6.3 Network Performance Results

Before we analyze the network for various ranges of parameter values, it is instructive to compare bus and network performance in small-scale systems (see Figure 10). As reported in the previous section, the Dragon scheme attains near perfect bus performance relative to the Base scheme for fewer than 16 processors and middle parameter ranges, while the Software-Flush and No-Cache schemes saturate the bus at 8 and 4 processors respectively. Because the network bandwidth increases with the number of processors, network performance becomes superior to buses when the bus begins to saturate. Both the Software-Flush scheme and the No-Cache scheme scale with the number of processors, though the Software-Flush scheme is clearly more efficient. The No-Cache scheme is poorer despite its smaller message size due to its higher request rate. Still, it scales with the number of processors and is a feasible choice if a designer wants to minimize hardware cost and software complexity. In a circuit-switched network, the request rate plays a more important role than the message size because of the high fixed cost of setting up the path to memory.

Because the network bandwidth scales with the number of processors (to first order), plotting processor

[5] We also estimated multiprocessor performance in a manner analogous to our bus analysis by representing the network as a load-dependent service center. The contention delay is computed using the queuing models described in [20]. This model gave virtually the same results as Patel's model.

utilization for a network of a given size is more interesting than in a bus-based system. Let us consider a network with 256 processors. Figure 11 shows processor utilization with various request rates for several choices of average message sizes. (Note that $2n$ must be added to the message size to get the network time per message.) Nine points are marked on the figure; they correspond to the performance of Base, Software-Flush, and No-Cache schemes with low, middle, and high parameter settings. The first letter in the label (B, S, or N) refers to the scheme, and the second letter (l, m, or h) refers to the range.

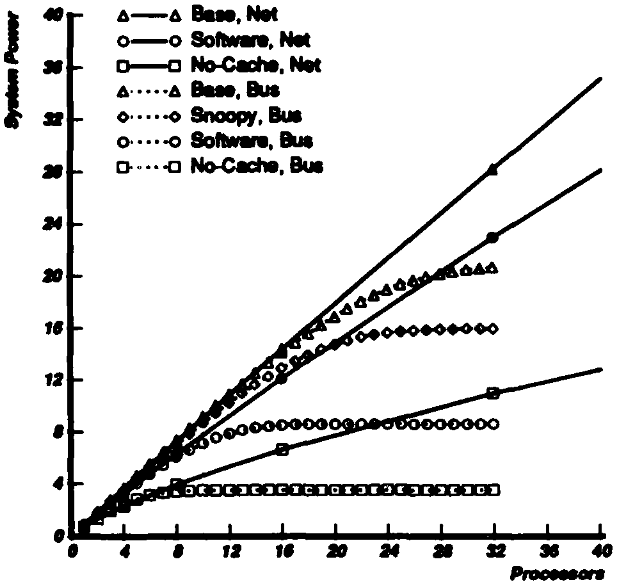

Figure 10: Buses versus networks in the small scale.

The first striking observation is the importance of keeping the network reference rate low. Even for a cache-miss rate as low as 3% in the 256-processor system and a message size of 4 words (corresponding to a unit-time service request rate of 3% × (16 + 4) = 60%), the processor utilization is halved. In a circuit-switched network, a change in the reference rate impacts system performance more than a proportional change in the blocksize. Of course, using a faster network, or using larger switches, will increase the reference rate at which the network latency begins to increase sharply.

The nine points fall into two performance classes. The Base scheme in all ranges, the Software-Flush scheme in its low and middle range, and the No-Cache scheme in its low range, fall into a reasonable performance category, and the other combinations are much poorer. While the Software-Flush scheme is usable even with medium sharing, the No-Cache scheme is efficient only if sharing is very low. Put another way, a system that does not cache shared data (and more so a system that does not cache *any* data) will need to use a much faster network relative to the processor to sustain reasonable performance. The performance of the Software-Flush scheme for the low range approximates the performance of hardware-based directory schemes. If Software-Flush schemes can attain a high value for *apl*, they have the potential to compete with hardware schemes in large-scale networks.

In the future we hope to examine reference patterns in large-scale parallel applications, both to get a better understanding of different workloads, and to validate our methodology against simulation. Traditionally, simulations have used synthetically generated traces, but a synthetic trace cannot capture workload-dependent effects such as hot-spot activity (or lack thereof) or locality of references. We are currently working on the generation of large multiprocessor traces for these studies. While we focused on a simple network architecture in this paper, we are interested in extending our results to other network

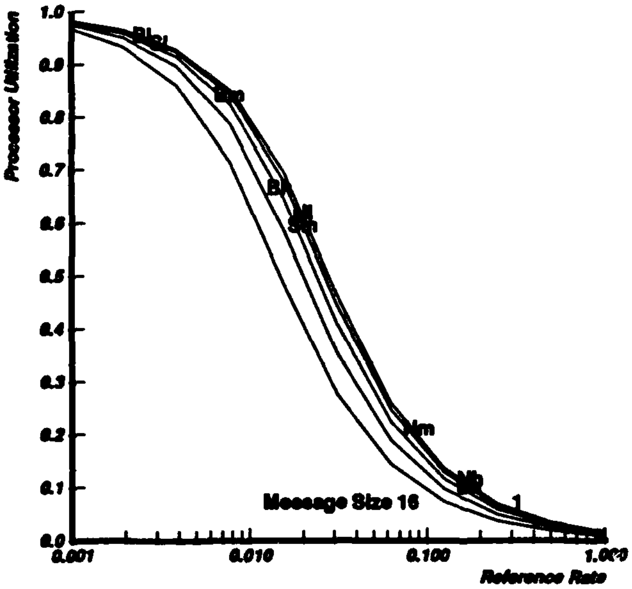

Figure 11: Bus performance for various request rates and with message sizes of 1, 2, 4, 8 and 16 words. The performance of the Base, Software-Flush, and No-Cache schemes is marked with two letter codes, the first letter (B, S, or N) corresponds to the scheme, and the second letter corresponds to a low, middle, or high (l, m, or h) range.

architectures, such as buffered and unbuffered packet-switching, as well.

7 Conclusion

Software cache-coherence schemes have been proposed and implemented because they have two advantages over typical hardware schemes: they do not require complex hardware, and they do not have the obvious scalability problem of a shared bus. However, to our knowledge, the performance of software-caching has not been analyzed before. In this paper we used an analytic model to predict caching overhead for several coherence schemes. The model was validated against multiprocessor trace data, and its sensitivity to variations in parameter values was studied.

First let us consider performance on bus-based systems. For almost all workloads, the snoopy cache scheme had the lowest overhead. Its performance was good for all workloads, while the software schemes showed great variation as the workload parameters changed. With a light workload (low memory reference rate and little sharing), the Software-Flush scheme was almost as good as snoopy cache, and even the No-Cache approach was feasible. Performance of the No-Cache method fell off dramatically as the workload increased. The performance of the Software-Flush method also deteriorated, though not as drastically.

We also evaluated the software schemes on a circuit-switched multistage interconnection network. Both software schemes scale well, as expected. Software-Flush does considerably better than No-Cache because it causes fewer memory requests, although the requests are longer. Use of packet-switching would be more favorable to No-Cache.

In both network and bus environments, the performance of Software-Flush is largely determined by the number of references to a block before it is flushed from a cache. This is a affected by program structure and by compiler technology. For example, the compiler can optimize performance by allocating related variables to the same block, and by flushing data as infrequently as possible. But if a shared variable is frequently updated by different processors, it is likely to have about two references per flush, no matter how sophisticated the compiler. At present we lack the workload data and compiler experience that would allow us to predict what is achievable here.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.

[2] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proceedings 1985 Int'l Conference on Parallel Processing*, pages 782–789, 1985.

[4] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, c-27(12):1112–1118, December 1978.

[5] H. Cheong and A. V. Veidenbaum. A Cache Coherence Scheme with Fast Selective Invalidation. In *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.

[6] David R. Cheriton, Gert A. Slavenberg, and Patrick D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 367–374, June 1986.

[7] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings ICPP*, August 1988.

[8] Jan Edler et al. Issues related to MIMD shared-memory computers: the NYU Ultracomputer Approach. In *Proceedings 12th Annual Int'l Symp. on Computer Architecture*, pages 126–135, June 1985.

[9] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.

[10] G. F. Pfister et. al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings ICPP*, pages 764–771, August 1985.

[11] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Up Memory Access Times. Electronics, 57, 1, January 1984.

[12] S. H. Fuller and S. P. Harbison. *The C.mmp Multiprocessor*. Technical Report, Carnegie-Mellon University, October 1978.

[13] G. R. Goke and G. J. Lipovski. Banyan Networks for Partitioning Multiprocessor Systems. In *Proceedings of the 1st Annual Symposium on Computer Architecture*, pages 21–28, 1973.

[14] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, June 1983.

[15] Albert G. Greenberg, Isi Mitrani, and Larry Rudolph. Analysis of snooping caches. In *Proceedings of Performance 87, 12th Int'l Symp. on Computer Performance*, December 1987.

[16] Norman P. Jouppi, Jeremy Dion, and Michael J. K. Nielsen. *MultiTitan: four architecture papers*. Technical Report 86/2, Digital Western Research Laboratory, Palo Alto, California, September 1986.

[17] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, June 1985.

[18] Clyde P. Kruskal and Marc Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers*, c-32(12):1091–1098, December 1983.

[19] D. H. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transactions on Computers*, c-24:1145–1155, 1975.

[20] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance*. Prentice Hall, 1984.

[21] E. McCreight. *The Dragon Computer System: An Early Overview*. Technical Report, Xerox Corp., September 1984.

[22] Steve McGrogan, Robert Olson, and Neil Toda. Parallelizing large existing programs - methodology and experiences. In *Proceedings of Spring COMPCON*, pages 458–466, March 1986.

[23] Robert Olson. Parallel Processing in a Message-Based Operating System. *IEEE Software*, July 1985.

[24] Janak H. Patel. Analysis of Multiprocessors with Private Cache Memories. *IEEE Transactions on Computers*, c-31(4):296–304, April 1982.

[25] Janak H. Patel. Performance of Processor-Memory Interconnections for Multiprocessors. *IEEE Transactions on Computers*, c-30(10):771–780, October 1981.

[26] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.

[27] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[28] Alan Jay Smith. CPU Cache Consistency with Software Support and Using One Time Identifiers. In *Proceedings of the Pacific Computer Communications Symposium*, October 1985.

[29] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[30] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[31] M. K. Vernon and M. A. Holliday. Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets. In *Proceedings of SIGMETRICS 1986*, May 1986.

[32] M. K. Vernon, E. D. Lazowska, and J. Zahorjan. An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols. In *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.